AUTOMATED TESTING WITH ACTS AND A MODEL CHECKER

This tutorial provides a step-by-step introduction to automated generation of tests that provide combinatorial coverage. Procedures introduced in this tutorial will produce a set of complete tests, i.e., input values with the expected output for each set of inputs. A “cookbook” style is used in the tutorial to keep the description concise, so it may not be intuitively obvious why certain steps are used. For the theory behind the processes introduced in this document, please see papers on the ACTS web site (http://csrc.nist.gov/acts). For questions on this tutorial, contact Rick Kuhn, kuhn@nist.gov.

To apply the methods described here you will need:

- The ACTS covering array generator and NuSMV model checker. ACTS is a free, open source tool developed by NIST and the University of Texas Arlington. ACTS is written in Java and can run in any OS with Java support. NuSMV, a variant of the original SMV model checker, is also freely available and was developed by Carnegie Mellon University, Istituto per la Ricerca Scientifica e Tecnolgica (IRST), U. of Genova, and U. of Trento. NuSMV can be installed on either UNIX/Linux or Windows systems running Cygwin. Links and instructions for downloading these tools are included in the appendix.

- A formal or semi-formal specification of the system or subsystem under test (SUT). This can be in the form of a formal logic specification, but state transition tables, decision tables, pseudo-code, or structured natural language can also be used, as long as you can write SMV code that matches the specification. The SMV code provides a precise, machine-processable set of rules that can be used to generate tests.

- An enumeration of values for each of the input parameters to be used in tests. The number of values per parameter should be limited to no more than 8 to 10. In most cases this will require establishing equivalence classes of parameter values. For example, values for an account balance may be 0, 1, 500, and 999999999 (or whatever is the maximum decimal value).

1 Overview

To apply combinatorial testing, two tasks must be accomplished:

1. Find a set of tests that will cover all *t*-way combinations of parameter values. ACTS will be used for this step. The covering array specifies test data, where each row of the array can be regarded as a set of parameter values for an individual test. Collectively, the rows of the covering array cover all *t*-way combinations of parameter values.

2. Determine what output should be produced by the SUT for each set of input parameter values. The test data output from ACTS will be incorporated into SMV specifications that can be processed by the NuSMV model checker for this step. In many cases, the conversion to SMV will be straightforward. The example in Section 2 illustrates a simple conversion of rules in the form “if *condition* then *action*” into the syntax used by the model checker. The model checker will instantiate the specification with parameter values from the covering array once for each test in the

covering array. The resulting specification is evaluated against a claim that negates each possible result R_j using a model checker, so that the model checker evaluates claims in the following form: C_i => ~R_j, where C_i is a set of parameter values in one row of the covering array in the form $p_1 = v_{i1}$ & $p_2 = v_{i2}$ & ... & $p_n = v_{in}$, and R_j is one of the possible results. The output of this step is a set of counterexamples that show how the SUT can reach the claimed result R_j from a given set of inputs.

The example in the following sections illustrates how these counterexamples are converted into tests. (Other approaches – covered in other tutorials – to determining the correct output for each test can also be used. For example, in some cases we can run a model checker in simulation mode, producing expected results directly rather than through a counterexample. In this tutorial we adopt a methodology that is consistent with mutation-based test generation procedures.)

The completed tests can be used to validate correct operation of the system for interaction strengths up to some pre-determined level *t*. Depending on the system type and level of effort, we may use want pairwise (*t*=2) or higher strength, up to *t*=6 way interactions. We do not claim this guarantees correctness of the system, as there may be failures triggered only by interaction strengths greater than *t*. In addition, some of the parameters are likely to have a large number of possible values, requiring that they be abstracted into equivalence classes. If the abstraction does not faithfully represent the range of values for a parameter, some flaws may not be detected by equivalence classes used.

2 Example

Here we present a small example of a very simple access control system. The rules of the system are a simplified multi-level security system, given below, followed by a step-by-step construction of tests using a fully automated process.

Each subject (user) has a clearance level *u_l*, and each file has a classification level, *f_l*. Levels are given as 0,1, or 2, which could represent levels such as Confidential, Secret, and Top Secret. A user u can read a file f if $u\_l \geq f\_l$ (the "no read up" rule), or write to a file if $f\_l \geq u\_l$ (the "no write down" rule).

Thus a pseudo-code representation of the access control rules is:

```
 if u_l >= f_l & act = rd then GRANT;
 else if f_l >= u_l & act = wr then GRANT;
else  DENY;
```

Tests produced will check that these rules are correctly implemented in a system.

3 SMV Model

This system is easily modeled in SMV as a simple two-state finite state machine. The `START` state merely initializes the system (line 8, Figure 1), with the rule above used to evaluate access as either `GRANT` or `DENY` (lines 9-13). For example, line 9 represents the first line of the pseudo-code above: in the current state (always `START` for this simple model), if $u\_l \geq f\_l$ then the next state is `GRANT`. Each line of the case statement is examined sequentially, as in a conventional programming language. Line 12 implements the "`else DENY`" rule, since the predicate "`1`" is always true. `SPEC` clauses given at the end of the model are simple "reflections" that duplicate the access control rules as temporal logic statements. They are thus trivially provable, but we are interested in using them to generate tests rather than to prove properties of the system.

```
   1. MODULE main
   2. VAR
        --Input parameters
   3. u_l:   0..2;         -- user level
   4. f_l:   0..2;         -- file level
   5. act:  {rd,wr};       -- action

        --output parameter
   6. access: {START_, GRANT,DENY};

   7. ASSIGN
   8. init(access) := START_;
        --if access is allowed under rules, then next state is GRANT
        --else next state is DENY
   9. next(access) := case
   10.     u_l >= f_l & act = rd : GRANT;
   11.     f_l >= u_l & act = wr : GRANT;
   12.     1 : DENY;
   13.     esac;

   14.     next(u_l) := u_l;
   15.     next(f_l) := f_l;
   16.     next(act) := act;

-- reflection of the assigns for access
-- if user level is at or above file level then read is OK
SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));

-- if user level is at or below file level, then write is OK
SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));

-- if neither condition above is true, then DENY any action
SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ))
         -> AX (access = DENY));
```

Figure 1. SMV model of access control rules

Separate documentation on SMV should be consulted to fully understand the syntax used, but specifications of the form "`AG ((`*predicate 1*`) -> AX (`*predicate 2*`))`" indicate essentially that for all paths (the "A" in "AG") for all states globally (the "G"), if *predicate 1* holds then ("`->`") for all paths, in the next state (the "X" in "AX") *predicate 2* will hold. In the next section we will see how this specification can be used to produce complete tests, with test data input and the expected output for each set of input data.

4 What Does the Model Checker Do?

Model checkers they can be used to perform a variety of valuable functions, because they make it possible to evaluate whether certain properties are true of the system model. Conceptually, the model checker can be viewed as exploring all states of a system model to determine if a property claimed in a SPEC statement is true. If the statement can be proved true for the given model, the model checker reports this fact. What makes a model checker particularly valuable for many applications, though, is that if the statement is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim in the SPEC statement can be shown false. The counterexample will include input data values and a trace of system states that lead to a result contrary to the SPEC claim (Figure 2).

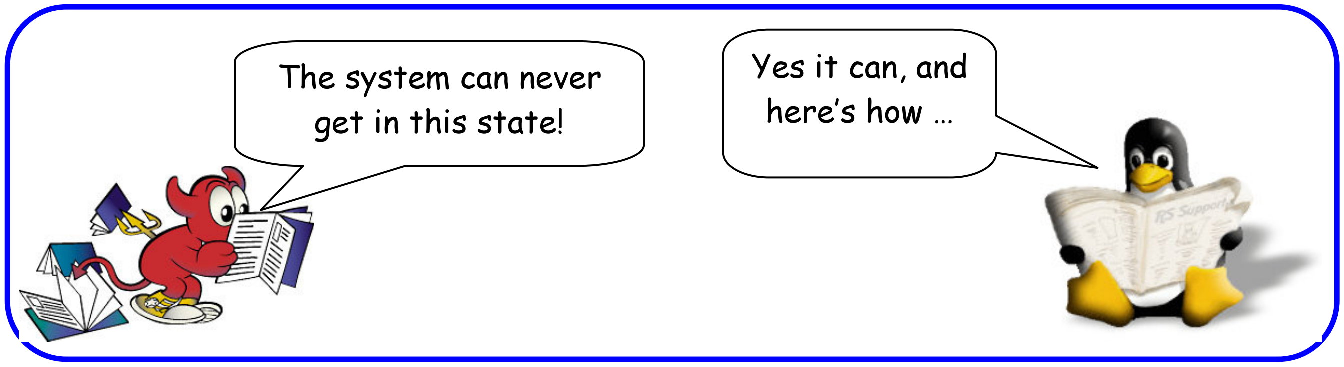
The system can never get in this state!
Yes it can, and here's how ...

Figure 2. Inside a Model Checker

For advanced applications in test generation, this counterexample generation capability is very useful. In this tutorial, however, we will simply use the model checker to determine whether a particular input data set makes a SPEC claim true or false. That is, we will enter claims that particular results can be reached for a given set of input data values, and the model checker will tell us if the claim is true or false. This gives us the ability to match every set of input test data with the result that the system should produce for that input data.

The model checker thus automates the work that normally must be done by a human tester – determining what the correct output should be for each set of input data. In some cases, we may have a "reference implementation", that is, an implementation of the functions that we are testing that is assumed to be correct. This happens, for example, in conformance testing for protocols, where many vendors implement their own software for the protocol and submit it to a test lab for comparison with an existing implementation of the protocol. In this case the reference implementation could be used for determining the expected output, instead of the model checker. Of course before this can happen the reference implementation itself must be thoroughly tested before it can be used as the gold standard for testing other products, so the method we describe here may be needed to produce tests for the original reference implementation.

5 NuSMV Output

Checking the properties in the SPEC statements shows that they match the access control rules as implemented in the FSM, as expected. In other words, the claims we made about the state machine in the SPEC clauses can be proven. This step is used to check that the SPEC claims are valid for the model defined previously. If NuSMV is unable to prove one of the SPECs, then either the spec or the model is incorrect. This problem must be resolved before continuing with the test generation process. Once the model is correct and SPEC claims have been shown valid for the model, counterexamples can be produced that will be turned into test cases.

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG ((u_l >= f_l & act = rd) -> AX access = GRANT)  is
true
-- specification AG ((f_l >= u_l & act = wr) -> AX access = GRANT)  is
true
-- specification AG (!((u_l >= f_l & act = rd) | (f_l >= u_l & act =
wr)) -> AX access = DENY)  is true
```

Figure 3. NuSMV output

6 Covering Array

We will compute covering arrays that give all *t*-way combinations, with degree of interaction coverage = 2 for this example. This section describes the use of ACTS as a standalone command line tool, using a text file input. A GUI based version of ACTS can also be used. Procedures for doing so are included in the Appendix. This example is small enough that the command line version is adequate and easy to use. The first step is to define the parameters and their values in a system definition file that will be used as input to ACTS. Call this file "in.txt", with the following format:

```
[System]
[Parameter]
u_l: 0,1,2
f_l: 0,1,2
act: rd,wr
[Relation]
[Constraint]
[Misc]
```

For this application, the `[Parameter]` section of the file is all that is needed. Other tags refer to advanced functions that will be explained in other documents. After the system definition file is saved, run ACTS as shown below:

```
java -Ddoi=2   -jar   acts_1.0b6.jar    ACTSin.txt
```

The "`-Ddoi=2`" argument sets the degree of interaction for the covering array that we want ACTS to compute. In this case we are using simple 2-way, or pairwise, interactions. (For a system with more parameters we would use a higher strength interaction, but with only three parameters, 3-way interaction would be equivalent to exhaustive testing.) ACTS produces the output shown in Figure 4.

```
Number of parameters: 3
Maximum number of values per
parameter: 3
Number of configurations: 9
-----------------------------
Configuration #1:
1 = u_l=0
2 = f_l=0
3 = act=rd
-----------------------------
Configuration #2:
1 = u_l=0
2 = f_l=1
3 = act=wr
-----------------------------
Configuration #3:
1 = u_l=0
2 = f_l=2
3 = act=rd
-----------------------------
Configuration #4:
     1 = u_l=1
     2 = f_l=0
     3 = act=wr
-----------------------------
```

```
Configuration #5:
1 = u_l=1
2 = f_l=1
3 = act=rd
-----------------------------
Configuration #6:
1 = u_l=1
2 = f_l=2
3 = act=wr
-----------------------------
Configuration #7:
1 = u_l=2
2 = f_l=0
3 = act=rd
-----------------------------
Configuration #8:
1 = u_l=2
2 = f_l=1
3 = act=wr
-----------------------------
Configuration #9:
1 = u_l=2
2 = f_l=2
3 = (don't care)
```

Figure 4. ACTS output

Each test configuration defines a set of values for the input parameters u_l, f_l, and act. The complete test set ensures that all 2-way combinations of parameter values have been covered. If we had a larger number of parameters, we could produce test configurations that cover all 3-way, 4-way, etc. combinations. ACTS may output "don't care" for some parameter values. This means that any legitimate value for that parameter can be used and the full set of configurations will still cover all t-way combinations. Since “don’t care” is not normally an acceptable input for programs being tested, a random value for that parameter is substituted before using the covering array to produce tests.

7 SPEC Claims with Combinatorial Test Values Inserted

The next step is to assign values from the covering array to parameters used in the model. For each test, we claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. (Note that the trivially provable positive claims have been commented out. Here we are concerned with producing counterexamples.)

Recall the structure introduced in Section 1: $C_i \Rightarrow \sim R_j$. Here C_i is the set of parameter values from the covering array. For example, for configuration #1 in Section 6:

```
u_l = 0 & f_l = 0 & act = rd
```

As can be seen below, for each of the 9 configurations in the covering array (Section 6), we create a `SPEC` claim of the form:

```
SPEC AG(( <covering array values> ) -> AX !(access = <result>));
```

This process is repeated for each possible result, in this case either “`GRANT`” or “`DENY`”, so we have 9 claims for each of the two results. The model checker is able to determine, using the model defined in Section 3, which result is the correct one for each set of input values, producing a total of 9 tests.

Excerpt:

```
...
     -- reflection of the assign for access
     --SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));
     --SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));
     --SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ) )
                 -> AX (access = DENY));

     SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = GRANT));
     SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = GRANT));
     SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = GRANT));
     SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = GRANT));
     SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = GRANT));
     SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = GRANT));
     SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = GRANT));
     SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = GRANT));
     SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = GRANT));

     SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = DENY));
     SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = DENY));
     SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = DENY));
     SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = DENY));
     SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = DENY));
     SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = DENY));
     SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = DENY));
     SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = DENY));
     SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = DENY));
```

8 Counterexamples

NuSMV produces counterexamples where the input values would disprove the claims specified in the previous section. Each of these counterexamples is thus a set of test data that would have the expected result of `GRANT` or `DENY`.

For each `SPEC` claim, if this set of values cannot in fact lead to the particular result R_j, the model checker indicates that this is true. For example, for the configuration below, the claim that access will not be granted is true, because the user's clearance level (`u_l = 0`) is below the file's level (`f_l = 2`):

```
-- specification AG (((u_l = 0 & f_l = 2) & act = rd) -> AX !(access =
GRANT))  is true
```

If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. In effect this is a complete test case, i.e., a set of parameter values and expected result. It is then simple to map these values into complete test cases in the syntax needed for the system under test.

Excerpt from NuSMV output:

```
-- specification AG (((u_l = 0 & f_l = 0) & act = rd) -> AX !(access =
GRANT))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  u_l = 0
  f_l = 0
  act = rd
  access = START_
-> Input: 1.2 <-
-> State: 1.2 <-
  access = GRANT
```

The model checker finds that 6 of the input parameter configurations produce a result of GRANT and 3 produce a DENY result, so at the completion of this step we have successfully matched up each input parameter configuration with the result that should be produced by the SUT.

9 Tests

We now strip out the parameter names and values, giving tests that can be applied to the system under test. This can be accomplished using a variety of methods; a simple script used in this example is given in the appendix. The tests produced are shown below:

```
u_l = 0 & f_l = 0 & act = rd -> access = GRANT
u_l = 0 & f_l = 1 & act = wr -> access = GRANT
u_l = 1 & f_l = 1 & act = rd -> access = GRANT
u_l = 1 & f_l = 2 & act = wr -> access = GRANT
u_l = 2 & f_l = 0 & act = rd -> access = GRANT
u_l = 2 & f_l = 2 & act = rd -> access = GRANT
u_l = 0 & f_l = 2 & act = rd -> access = DENY
u_l = 1 & f_l = 0 & act = wr -> access = DENY
u_l = 2 & f_l = 1 & act = wr -> access = DENY
```

These test definitions can now be post-processed using simple scripts written in PERL, Python, or similar tool to produce a test harness that will execute the SUT with each input and check the results. While tests for this trivial example could easily have been constructed manually, the procedures introduced in this tutorial can, and have, been used to produce tens of thousands of complete test cases in a few minutes, once the SMV model has been defined for the SUT.

APPENDIX

ACTS download and install

Email me if you don’t already have it (we like to keep track of the number of users; helps maintain management support!): kuhn@nist.gov.

NuSMV download and install

http://nusmv.irst.itc.it/

Script to generate tests from SMV output used in Section 9 for the example

Input: output file from NuSMV
Output: tests.txt file of test definitions described in Section 9

```
# produce tests from smv output
grep specification smvoutput | grep "is false" \
|sed 's/(//g' | sed 's/)//g' >tests.txt
ex tests.txt <<END
:1,$ s/--.*AG//
:1,$ s/AX !//
:1,$ s/is false//
:w
END
```